mcode primer

mcode primer, project notes and implementation notes
7/24/2024

About this document

This document (currently about 50 pages) is intended to be a primer
for the new 'mcode' language, an idea repository, a discussion of design
decisions made, and to act as a source for project spin-off ideas.

    nb. There may be some repetitions in this document since
    it is a work in progress.

    nb. If you are reading this in the 'mcode viewer' or in the
    'mcode console' then you will see syntax highlighting.
    You can also read or print the mcode source
    and documentation as files (but without syntax highlights).
    The highlight colors do help comprehension quite a bit.

        Changes you might make in the 'mcode console' are
        -not automatically saved-

    nb. There are many sidebars and digressions. This document will
    be re-organized into spin-off papers on concepts, implementation,
    and applications that will be more focussed.

    nb. N codes ...
        NYI is not yet implemented,
        NYC is not yet completed,
        NIU is not in use,
        NFN is non-functional (broken, needs repair),
        NDY is not determined yet.

mcode Virtual Input Keyboard (VIK) and IDE

    The Integrated Development Environment (IDE) is started by either
    loading 'mcode.html' in a web browser,
    or running the 'mcode' desktop application (made with Neutralino).

    Since mcode is a transpiler, all the features of the native JavaScript
    debugger in the web browser are available.

    A virtual input keyboard is provided in the IDE for input of symbols:
    /// notebook vik1 14 img lib/vik.png

    The IDE and VIK are intended to be usable on mobile and desktop devices.

    If a physical keyboard is in use, then symbols can be input be using a
    ` prefix key.

        Examples:
            `a inputs α
            `S inputs Ϋ

    VIK is defined in the 'core' file for mcode, and is written in mcode.

```
    To make changes, edit 'core' and press Regen in the IDE.

    Some symbols are shared with Dyalog and GNU APL languages,
    and many are new to mcode.  Differences are listed below in this document.


mcode Primer and Early Implementation Notes
-------------------------------------------

mcode is a 'meta hybrid' language, implemented at this time as a
transpiler in JavaScript.

    meta - mcode is a language that can be used to reason about other
        languages
    hybrid - mcode is a language that is 'open' and can include parts
        of other languages to allow mcode based software to be built
        incrementally and easily.

mcode is intended to:
    - develop algorithms and code in a concise mathematical way:
        rapidly, with less error, and in a math based notation rather
        than using a more traditional keyword based language
    - make web development faster and simpler (for the JavaScript
        version) nb. the main interest of the co-author (David)
    - encourage code noodling / experimentation and
        fully support development of software comfortably on mobile
        devices
    - enable development of rich media 'notebooks' that include graphs,
        diagrams, equations, and media.  TBD: export to static HTML.

            nb. The notebook capability was added to the ACE editor
            (Advanced Code Editor) which is similar to Microsoft
            Visual Code.  ACE notebooks are simply embedded or
            generated HTML or images.

            nb. Comparison to Jupyter notebooks: similar, but fully
            cloud/web based, maybe simpler: there is no need for
            Docker or desktop executable downloads.

mcode can be transpiled to JavaScript (JS) at this time.
Python, and other targets may be planned, such as MATLAB, or Julia.

mcode was inspired by the APL language, and is functional, generic, and
array oriented, and could be vectorized for numeric computing on the GPU.

    sidebar: APL has been vectorized in the co-dfns project.
    see: https://github.com/Co-dfns/Co-dfns
    This work was a PhD thesis by Aaron Hsu, demonstrating a
    parallel compiler (running on the GPU) of an array based language.

Currently, input of non-standard symbols can be made using "apl_vk"
which is an open-source virtual keyboard HTML GUI widget for desktop
and mobile devices. This is currently integrated into HTML Textarea
and the open source Advanced Code Editor (ACE).

    see https://ace.c9.io/
    an mcode input widget is at: https://primos.family/daver/apl_vk_demo


sidebar: mcode and other computer languages
```

A Python transpiler would be a highly useful target.
Since the transpiler is roughly 1/2 table driven, by using a
JavaScript to Python converter or AI, an mcode Python transpiler may not
be difficult to implement. Certain concepts may yet have to be remapped,
such as the tuple datatype (which is not universal).

Other languages such as PHP may not be of academic interest, but have been
-practical- for web applications especially for server side code.
Therefore an mcode to PHP transpiler might be of use.

mcode would probably not be very helpful for highly complex syntax
languages like Java or C++ (which have many reserved words) although mcode
representation may be of interest, particularly in the area of generics
and algorithms.

There are many interesting, more 'exotic' languages, such as Haskell,
Standard ML, eiffel, elm, DART, Go, Scheme, and languages with rich history
still in use such as FORTRAN, C, and LISP. There could be interesting
research in the area of mcode ideas applied to these languages perhaps
as interpreters, or code generation from algorithm specification.

Introduction

- mcode is an open language
- mcode itself is built incrementally
- mcode is written in mcode as far as possible.

Also, mcode does not run in it's own virtual machine,
but rather in the target language ecosystem.

Since the languages can be mixed (to some extent) the target is referred to
as the 'underlying lanugage'. If the target language is compiled only,
then mcode can be used as a preprocessor for the compiled language.

mcode is intended to be a notation, and is not necessarily oriented
towards specific interpreters or compilers. However, it may be most
suited as a 1) a notation for pseudo-code or 2) a front end for
procedural languages, and 3) as an adjunct for Database or Domain
Specific Languages, and 4) a meta-language or notation for
representation of algorithms for AI/ML systems.

A use case under consideration is mcode as a notation in
conjunction with machine learning for translation and represention
of algorithms and methods, to/from many computer languages.

There may also be value for mcode as a concise notation for
stored procedures in databases and in
model-based systems engineering (MBSE).

The themes of mcode are:
    - be concise like mathematics
    - minimal punctuation for clarity
    - indendation and spacing to clarify program structure
    - array oriented as much as possible
    - functional, using other functions as data sources / sinks

Users can define more primitives and add them dynamically.
This is how the 'core' script builds up the language.
(this is usually pre-built and cached in the underlying language)

A Taste of mcode

- mcode symbol examples:

statements	∇	define function
primitives	∈	is an element of
vector/matrix	[ ]	contains data or expressions
algebraic grouping	( )	changes the order of execution
lamba/anonymous	{ }	defines an unnamed function in-line
comments	//	1-line comment, also /* for multiline */

nb. syntax highlighting with the 'cobalt' theme in ACE:

statement symbols	∇
primitives	∈
built-in symbols	+
grouping symbols blue	( )
comments	//
strings	'string'
numbers	3.14
mcode or _ library	_.someFunction
an error is reported as	error: message

- lines are evaluated top down ↓

- expressions are evaluated right to left ←

- there is no operator precedence, use ( ) to group operations if needed

- there must be spaces between primitives, data, and grouping

- if no spaces then the entity is passed to the underlying language

- indents control automatic insertion of { }

Examples:
---

reading guide:

⍳	create a vector 0..⍵
{ }	define a temporary / lambda function with left arg ⍺ and right arg ⍵

```
> ⍳ 8
[ 0 1 2 3 4 5 6 7 ]
> 2 ** ⍳ 8
NaN
// NaN since exponentiation (power) is built-in and not vectorized (yet)

// using 'each' to vectorize the built-in (scalar) function **
> { 2 ** ⍵ } ¨ ⍳ 8
[ 1 2 4 8 16 32 64 128 ]
// make more general: provide the base as argument ⍺
> 2 { ⍺ ** ⍵ } ¨ ⍳ 8
[ 1 2 4 8 16 32 64 128 ]
// calc vector of bases ⍺ raised to ⍵, return result as [⍺,⍵,⍺**⍵] vectors
> U = ⍳ 2
[ 0 1 ]
> V = ⍳ 3
[ 0 1 2 ]
```

```
> U { [α,ω,α**ω] } ¨ V
[ [[2,0,1],[2,1,2],[2,2,4]] [[3,0,1],[3,1,3],[3,2,9]] ]

    nb.  some commas are converted to space to improve readability
    use ⍎ format to get a JSON version of the output:

> ⍎ U { [α,ω,α**ω] } ¨ V
[[[0,0,1],[0,1,0],[0,2,0]],[[1,0,1],[1,1,1],[1,2,1]]]


More Complex Example: define the primitive 'each', which was used above

    reading guide:

    ∇       define function named ω
    ⍜       for (iterate over vector/list)
    →       if α then ω
    ◇       else
    ⍛       return
    ↓       push
    ⍳       create vector 0..ω
    b,r     local variables
    []      empty vector
    α       left argument in any function
    ω       right argument in any function
    δ       'modifier' to any function as in op.δ


input to define a primitive ¨ called 'each'
-----
∇ mcode.each : d,r=[]                   // ¨    apply function α to each element in vector
ω
    ∇ w : b,s=[]
        b ⍜ ω : s ↓ δ α b ; ⍛ s
    δ ∈ [] →                            //     L arg for α is in δ
        d ⍜ δ : r ↓ w(α,ω,d)
    ◇ r = w(α,ω,δ)
    ⍛ r
'¨' ∆ mcode.each


    nb. the mcode JavaScript IDE is a text editor with a
    read-eval-print-loop (REPL).  The prompt is >

    In the IDE the last input line is executed, or if a block of code
    is selected that is executed.


output for ¨ 'each' JavaScript from transpiler
------
mcode.each = (_a,_w,_d) =>  {           // ¨    apply function α to each element in vector
ω
    let d,r=[]
    w = (_a,_w,_d) =>  {
        let b,s=[]
        for (b of _w) mcode.push(s,( _a(_d,b,null) ),null); return s
    }
    if (mcode.typeof(_d,'[]')) for (d of _d) mcode.push(r,w(_a,_w,d),null);
                                        // L arg for α is in δ
    else r = w(_a,_w,_d);
    return r
```

```
    }
    mcode.create('¨','mcode.each',null);
```

Web Application Use Case or Why mcode?

The co-author (David) was motivated to start this project to make development of web applications simpler and more enjoyable, and to use array oriented and functional programming styles.

mcode for web development at this time mainly focuses on JavaScript, but may address HTML and CSS components as well.

There are several APL systems and several interesting array/functional programming languages available. Most are either implemented as standalone systems, or as a virtual-machine on a virtual-machine. Also, none had strong integration with the web page Document Object Model, DOM and the web browser.

alternatives considered included:
- using a proprietary closed source software product, Dyalog APL
- using a mature open source APL, such as GNU APL
- using a community grass-roots APL, such as a Java or JavaScript APL

mcode had web development as its primary initial target.

The mcode spacing rule

The initial authors decided that by:

- designing and implementing a simple and invariant expression parser, always using a triplet structure: Left Op Right some of the qualities of an array based and functional language like APL could be realized.

    nb. a complex Abstract Syntax Tree (AST) was not used to implement the expression parser, instead the AST consists of nested expression triplets.

- using a very simple lexer that is -space sensitive-, mcode triplets could be separated from an underlying language, and thus not -all- of the requirements of a working language needed to be built initially.

    - some readers will dislike the spacing rule in mcode. Stop reading. Go pound sand.

An example of the spacing rule:

```
0 ∈ [ 0 1 2 ]   is valid, the vector is in mcode format with spaces

0 ∈ [0,1,2]     is valid, the vector is valid JSON
```

nb. since the vector has no spaces, it is treated as a single token in the mcode parser.

```
0∈[0,1,2]       is -invalid- mcode but is still
```
passed to the underlying language (interpreter, compiler,

```
        REPL, etc, depending on the ecosystem), which for JavaScript
        will report:
             error: Unexpected 'ϵ'. E024
             stopping report
        Attempting JavaScript evaluation and execution:
        SyntaxError: Invalid or unexpected token
        during JS eval of:
        return 0ϵ[0,1,2]; ...

        nb. The first error is from the 'jshint' code checker.
        The second error is from the JavaScript evaluation.

    There were two advantages to using mcode spacing rule:

    a) it made development of the lexer much faster and simpler

    b) at this time a formal EBNF formal specific of mcode has not
    yet been written, since mcode syntax was adjusted/developed
    -during- the development of the transpiler and of the core
    implementation (file mcode.mc).

- spaces are allowed in data, such as strings:

    for example:
    > 'a b c' ϵ [ 0 'a b c' 2 ]
    1
    > [ 3 'a b c' ] ϵ [ 0 'a b c' 2 ]
    [ 0 1 ]

- since spacing was already a part of mcode, -indentation- as used in
Python was also made part of mcode.  Any indented code is wrapped in
{ }

This leaves explicit writing of { } free to be used for 1-line lambda
(anonymous) functions, which are very useful for asynchronous or
i/o completions, and GUI callbacks.

{ } are not required in mcode to define functions or data structures,
as in JSON, but -can be used with no spaces.-

For example, below is a valid mcode session in the JavaScript
ecosystem:

mcode
-----
let_ d={'keys':['a','b','c'],'values':[0,1,2]}  // let_ passed to JS
 d                                              // print d

output of transpiler
--------------------
let  d={'keys':['a','b','c'],'values':[0,1,2]}; // let_ passed to JS
mcode.quad('d',d); // print d

runtime console outout
----------------------
d = { keys ["a","b","c"]  values [0,1,2]  }

    nb. JavaScript allows ' or " or ` to enclose strings.
```

sidebar: the 'dog food' idea

```
    The 'dog food' idea is used here in that mcode is
    used to build mcode. (eat your own dog food and enjoy it)

        Early LISP was built-up in much the same way.
        Only a few key atomic data types and operations were written
        in assembler language.  In college, author (David) loaded
        9 track tapes on the IBM System/360 and was fascinated
        as the language built itself up.

    In modern times, see for example, the Emacs embedded LISP
    interpreter:
    https://github.com/emacs-mirror/emacs/blob/master/src/eval.c


sidebar: mcode / JavaScript transpiler development stats

    The mcode transpiler is about 1500 lines of JavaScript in
    these modules:

        init, lexer, expr, codeCheck, parser, mexec, coreLoader

        as well as functions for logging and simple data analysis

    These are implemented as named functions to provide local variable
    closures to keep the code organized (and in JavaScript functions
    act like classes)  For native mcode transpiliers for other languages
    these modules would be declared as classes.

    Time and effort:  ideas for this project percolated for a few
    years with only experimantal code fragments written (this was called
    apl0js).  Other approaches were investigated, such as using Dyalog
    (closed source, proprietary), GNU APL (open source, quite large),
    and other open source projects.

    In April 2023 the idea for a triplet expression parser was implemented.

    By October 2023 this was completed:
        - working mcode transpiler (file mcode.js)
        - partial core primitive set (file core)
        - integration with the Advanced Code Editor

    An interface to ACE was also completed, with syntax highlighting,
    indenting, a console, and file load/save/on to the cloud.
    ACE is used by the Amazon Cloud9 IDE.

        nb. Microsoft Visual Code editor, Monaco was also considered, but
        ACE seems to have a better adoption over the web with many products
        and platforms using it.  Overall it is stable and works as expected.

        MS VC uses a more complex Language Server Protocol for syntax
        highlighting. Dyalog uses the MS VC editor (called Monaco), for the
        open source RIDE (remote IDE) for Dyalog APL.
        see: https://github.com/Dyalog/ride


sidebar: Web Page Notation Proposal

    A project is underway to develop a simpler, alternative notation
    for certain cases of HTML/CSS/JS:
```

```
    HTML is a markup language in the XML family and
    requires the <X> ... </X> syntax which could be made more concise.
        - for static documents with mostly text,
          MarkDown seems a good solution
        - for dynamic documents, a syntax hybrid of Emmet and TCL/TK could
          be a good direction

        - define child elements from indentation or naming:
            top
                form
                    label
                        entry
            or
            top.form.label.entry     // creates the elements above entry also
                                     // autoname, autoclass (idea from TCL/TK)

        - syntax for leading character for
            HTML-elem .class #id attr=value
            CSS in a separate file for syntax checking and with autocomplete
            mcode callbacks


The Dataflow Idea

    In mcode expressions, data flows always from right to left ←

    Example, if x y z are data items, and F and G are functions, we
    can write:

    mcode               meaning
    -----               -------
    y F x               F(x,y)
    z G y F x           G(z,F(x,y))

    This example shows how functions are composed.

    The rightmost element must be data.  To invoke a call to F
    with no data or unknown data we can write:

    F 0                 F(0)

    F alone means data, a variable or object called F containing
    or referring to data.

    object oriented programmming is supported.
    Functions can be methods of objects.
    mcode introduces some symbols, for example:

    Δ                   self  or  this
    ⊙                   shared data object


mcode and ML / AI tools for software development

    Below is an outline for proposals on mcode with AI/ML and MBSE projects:

    * improve the breadth, accuracy and usefulness of existing
      coding tools; eg. similar to "Copilot for GitHub"

        - by using mcode as a lingua franca intermediate
          representation for algorithms and methods
```

```
    * also improved code generation automation tools, eg. from model
      based system engineering (MBSE) models to usable code

        - by storing concise mcode inside the MBSE model directly
          use case for SW development

    * developer working in C++ but good algorithm examples
      pertaining to their task at hand are only available in Python

    * solution:

        - software developer prompts AI model which finds mcode
          example(s), and the response is automatically translated
          from mcode to useable and correct C++ in real time

    project picture:

    * learning phase: translate existing code base (a huge DB) to mcode,
      as a common language, a lingua franca

    * tool development phase: use ML system to develop a software development
      AI model to drive a coding assistance toolset

    * convergence/validation phase: compare correctness of generated
      responses to known existing and valid code bases (libraries)

    * tailoring phase: validate translations from mcode intermediate to
      several specific computer languages

    more:

    - concept: translate existing code base (a huge DB) to mcode,
        as a common language, a lingua franca;
        then: ingest and train ML system and develop a software development AI
            model to drive a coding assistance toolset
        - AI responses can be tailored back from mcode as an intermediate to
            a specific language
        use case:
            - developer working in C++ or Rust, but good algorithm examples
                pertaining to their task at hand are only available in Python
            - solution:
                software developers query (at the time) finds mcode example,
                and the response translated from mcode to language of need
                at the moment
    - could improve the breadth, accuracy and usefulness of existing
        AI coding tools
    - could lead to improved code generation automation tools


mcode and APL

    mcode is not APL, but is influenced by, and shares some traits with APL.

    APL is a terse array-oriented interpreted language.  More recently
    it has been implemented as open source for the web and in the Java.

    Dyalog Ltd is the leading vendor of a commercial APL product used
    mainly the insurance actuarial and finance industries.

    If you are interested in APL, there are two main versions.  These
```

are full modern implemenations that run on the desktop and Raspberry PI:
    https://www.dyalog.com/
    https://www.gnu.org/software/apl/

There is an open source Java VM APL also:
    https://github.com/dzaima/APL/tree/master

An excellent book on array programming in APL is:
    https://www.dyalog.com/mastering-dyalog-apl.htm

There is an APL wiki here:
    https://aplwiki.com/

The ideas of functional and array programming are implemented in newer languages such as R and Julia.

Symbol and Layout differences from Dyalog and GNU APL

    Changes compared to Dyalog:

    adds:

      q ∞ ⍰ w ⍵   r ⍲   y ⍐   u ⍗
      a ⍺   s ⍫   d ⍛   f ⍥ ⍤ g ⍢   h ⍙   k ⍟
      x ⍉   c «   v »   b ⍪   n ⍀   m ‖   . ⍁

    moves:

      ! on - to ⍘
      ⍁ on ` to ⍥ (tbd)
      ⍙ on . to h

    changes:

      < on 3 to ⍃
      > on 7 to ⍄
      = on 5 to ≈
      ∨ on 9 to ⍲

      ⍫ on (to ∨
      ∧ on 0 to ⍬
      ⍱ on) to ⍷

      * on p to ⍴

      | on m to ⍞

    more single unicode symbols in the APL386 font for consideration:

    ± ⍞ « » € ⍲ ⍥ ≈

    ⍺ ⍛ ⍬ ⍵ ⍜ ⍹ ⍵ ⍤

    ↭ ⇞ ⌀ ⍲ ⍇ ∑ ∫

    ∏ ⍴ ∏ ⍷ ⍷

    ‖ ⍘ ‖

∉ ⊔ ⊙

∞ ⊇ ⌦ ⌫

⍂ ⍃ ⍄ ⌸ ⌻ ⍰

⍶ ⍹ ⍊ ⍑

⍔ ⍍ ⍙ ⍐ ⍗ ⍢

⍡ ⌾ ⍭

╫ ⌽ ⍛ ⍜ ╙

⊗

∠

for more:
use MS Word, font APL385, insert symbols

see FontDrop
https://fontdrop.info/#/?darkmode=true

see:
http://xahlee.info/comp/unicode_APL_symbols.html
http://xahlee.info/comp/unicode_math_operators.html

mcode Language Overview

mcode is structured into functions, statements, expressions, data, comments.

nb. The underlying language provides facilities for classes, complex data types, and data literals, such as JSON.

Functions contain statements, and these statements are indented (as in Python). Each indent level requires at least two spaces.

mcode symbol examples:

statements like	∇	define function
primitives like	∊	is an element of
vector/matrix	[ ]	contains data or expressions
algebraic grouping	( )	changes the order of execution
lamba/anonymous	{ }	defines an unnamed function in-line
comments	//	1-line comment, also /* for multiline */
comments	# ⍝	Python style and APL style 1-line comments

Between these symbols there -must- be a preceding and trailing space.

mcode expression parts are spaced, and expression parts for the underlying language are not spaced.

nb. orginally this spacing rule was done to keep the lexer code very simple, however a useful finding was that non-spaced input could be treated as a single token and processed by the underlying language, thus enabling mcode to be an -open- language. This will be explained below.

```
For example:
    input                           note
    -----                           ----
    ∇ fn                            defines a function named fn
        ...                         some code
        ⍒ r                         return statement

    ∇ fn : x=0,y=1                  function with initialized local vars
        ...                         some code
        ⍒ r                         return from function with result r


        nb. : is often a separator betweem statement clauses, as in Python


Within a function special symbols refer to the Left and Right arguments:

    α   Left argument, or null if none available
    ω   Right argument, cannot be null

Statements contain one or more expressions.
Statements are processed top down, and left to right.

An Expression is always formed as:

    L op R

where L can also be an op or ( ) or { }

'op' is short for 'primitive' which are the symbols that become
function calls when transpiled.

Primitives form the language, and this is a concept from the APL language.
They are usually one symbol (or rarely two symbols).

( ) change the order of evaluation from the normal Right to Left
and must also contain L op R groups or op R.
The furthest right entity of a statement must always be data.

+ 0     // valid
0       // valid

{ } define an anonymous function (also called a 'lambda'), explained below.


Expressions are processed right-to-left ←
For example:

input                           output
-----                           ------

2 ∈ [ 0 1 2 ]                   mcode.memberof(2,[0,1,2])

    nb. mcode.memberof is a function in the mcode runtime library

x F y                           F(x,y)

    in a cannonical form we can write:
    α op ω                      op(α,ω)
```

```
        nb. op refers to any symbol called a 'primitive' or a named function

    A few more examples, using functions op1 and op2:

    0 op1 1 op2 2                   op1(0,op2(1,2))

    ( 0 op1 1 ) op2 2               op2(op1(0,1),2)

        parentheses alter the data flow
        parentheses must be spaced
        L is null for op2

    0 op1 [ 1 2 3 ]                  op1(0,[ 1, 2, 3 ])

        lists use space delimiters as in MATLAB


    Logging and Pass-Thru Primitives

    □ ω                 logs ω to both JavaScript console and IDE session log
    α □ ω               logs 'α' = ω

    α ⊢ ω               pass-thru or not-yet-imlemented, logs α and ω as in □

        nb. □ symbol is named 'quad'   It also performs other functions,
        as in APL.  □.src 'fn' returns the source code of a function named 'fn'


Infix Notation vs Prefix Notation

    Mathematics uses both notations:

    F(x,y)  is prefix notation
    x+y     is infix notation

    mcode always uses infix notation, and in groups of 3 as:

    L op R

    if L is not given then it is null

        For comparison, LISP and most shells uses prefix notation, as in:

        command-word parameter-word-list
        F            x y
        +            x y

    The mcode equivalents are:

        x f y
        x + y

    In cases where a functions requires more than 2 arguments, there are
    several options:

        1)  provide a vector argument for L or R, for example:

            x F [ y z ]

        2)  provide an object argument for L or R, for example:
```

```
        x F object

            where F then accesses elements inside of 'object'

    3)  define F to be a method of a class or object which can access
        more data

        x object.F y

            where F also has access to data by using 'this'

    styles 2 and 3 above combine object oriented data access with
    dataflow access.


Also, since mcode is a hybrid language, the syntax of the underlying
language can be mixed in:

    an immediate object (structure) can be an argument for L or R

    x f {x:1,z:2}

        note that {...} is JSON and may be specific to JavaScript

One can write a prefix function call listing the arguments.  The exact
syntax of this call depends on the underlying language and may support
default arguments and other features.

    f(x,y,z)


Hybrid code mixed with mcode expressions

mcode grammar allows mixing of the underlying language with mcode
without any need for escapes, quotes, or special function calls.
A simple rule is used, and may work well in languages that are
not space sensitive:

mcode expressions are spaced, and expressions in the underlying language
are not spaced.

f(x,y)              not spaced, seen as one token

x f y               spaced, seen as L op R

The implication of the spacing rule is that the grammar of mcode is
not closed, and it is extremely simple and consistent, always formed
of expression triples, where L is optional:

    L op R          expression triplet, L optional, op is a 'primitive'
    ( ... )         expression grouping

An expression triple is always converted to
a function call.  This is infix to prefix converion, as shown above.

mcode also includes:

    [ ... ]         vector or list data     ... are elements in underlying language

                    [ 1 2 3 ]
```

```
                  [ 1 object.var ]

    { ... }       lambda function        ... is an mcode 1 line expression

                  { ⍺ + ⍵ }

    ∇ ⍐ ⍗ ⍈       statement symbols, some listed here, explained below
```

Any tokens that are not spaced are passed to the underlying language, after some substitutions for variables and constants, like ⍺ ⍵ ⍿ etc.

A single token that is unspaced is seen by mcode as 'R' which is data. No function call is generated, and it is passed to the underlying language. In the transpiler this is called a 'solo' expression.

There is also a syntax for matrix specification, for example:

```
M ← [   1,  0,
        0,  1   ]
```

Commas are required after each matrix element which can each include an mcode expression.

```
Assignment  =  vs  ←

M ← [   1,  0,
        0,  1   ]
```

In the example, the matrix initialization is a multiline statement, therefore a different syntax from assignment was decided upon. To be consistent with mcode, special symbols are used for statements, in this case ←

← is also used for 1 line named function assignments like:

```
    foo ← { ...mcode... }
```

These symbols like ∇ ← etc. are 'out of band', meaning they don't conflict with reserved keywords or other entities in underlying language.

For matrix definition the symbol is ←.
Note that each term in the matrix, which is separated by comma space, may contain valid mcode or non spaced expressions in the underlying language.

= is more complex. = will have the syntax and semantics as defined by the underlying language, which is almost always assignment.

However, = may be used in expresions, such as:

```
    mcode       output                      console
    -----       ------                      -------
    ⍞ 1+(x=1)   mcode.quad(null,1+(x=1));   2
    x           x                           1
```

nb. prints to the console, and can also do other things.

nb. global var x was created (per the rules of undeclared variable assignment in JavaScript)

Matrix Indexing

Individual elements in a matrix may be accessed as:

M[i,j]

Example:

```
> M = [ 4 4 ] ρ ι 16
[   0,  1,  2,  3,
    4,  5,  6,  7,
    8,  9, 10, 11,
   12, 13, 14, 15  ]
> M[2,3]
11
```

nb. indices start at 0

Casts and Direct Function Calls

```
foo( x,y )
```

will be passed directly to the underlying language.
This supports casts and direct function calls.

Spacing rule examples:

Function call to 'foo':

mcode	result	note
foo(x,y)	foo(x,y);	single token pass through
foo(x,y)	foo(x,y);	called as 'op R'
foo (x,y)	foo(null,(x,y),null);	called as 'L op R' and L is null

Cast to Number:

mcode	result	note
Number(x)	Number(x);	valid
Number(x)	Number(x);	valid
Number (x)	Number(null,(x),null);	* called as Number(α,ω,δ)

* since Number is a reserved constructor in JavaScript
this is invalid code.

Cast to Number with mcode primitive expression:

	mcode	result	
	Number(x⊢0)	Number(x⊢0);	-invalid-
>	Number(x ⊢ 0)	Number((mcode.memberof(x,0,null)));	-useful-
	Number (x ⊢ 0)	Number(null,((mcode.memberof(x,0,null))),null);	

The expression marked > is useful. The other two will generate

evaluation errors.

Writing mcode

Unary (Monadic) Primitives

Some operators in various languages have different functions depending on the form: L op R or op R

	mcode/APL terminology	
op R	Monadic	op is unary
L op R	Dyadic	

to write expressions that force op to be monadic, write:
op1 (op2 R)

op2 is forced to be monadic.
op1 get the result of op2 R

For example:

mcode	result	note
foo - b	foo - b	foo treated as data
foo(- b)	foo((- b));	foo treated as direct call
foo (- b)	foo(null,(-b),null);	foo treated as mcode function

mcode functions

User written functions can be directly incorporated into mcode expressions.

The signature for a function that will be called by the transpiler is:

Left function Right or α op.δ ω

nb. .δ is optional

op(α, ω, δ)

where: α is L or null ω is R

δ is the 'modifier' for 'primitives' (explained below)

Function arguments α ω δ

in mcode ω is converted to _w, α to _a, δ to _d

in an mcode triplet 'L op R'
α maps to L, ω to R, δ to a modifier
the modifier is the .X after a primitive, as in 'L op.X R'

within an mcode function the symbols α ω and δ are local variables
mcode functions should always return a value, ω if the function does not modify the incoming data, but instead has 'side effects'

mcode promotes Generic Functions

The idea of DUCK typed data in many scripting languages is taken further in mcode to allow operations that are similar in purpose to perform a reasonable or expected operation on data. This is also called 'generics' in most computer languages. In mcode, primitives are as generic as possible and reasonable.

For example ϵ (member of) can operate on strings, vectors, maps, and tables. In languages like JavaScript or Python widely varied functions and syntaxes are often required to perform this operation.

nb. In C++ for example, this idea is implemented with templates with quite complex semantics. For example, the standard library <algorithm> defines a binary search, and then at -compile time specific code is generated- to implement binary search for the -exact- input and output data types. This gives an efficient performance solution by the use of a template sub-language within.

see:
https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/group__binarysearch.html

In some other languages, like Java and C# this idea is implemented with first class 'generics'. Generics handle differing datatypes at runtime, whereas templates generate precompiled variants of a prototype algorithm at compile time.

Macros are another way to varying generate code at runtime, often with #if compiler directives.

Macros are also notably a part of LISP and its derivatives.

nb. April (Array Programming Re-Imagined in Lisp) is a compiled APL implementation embedded in Common Lisp.

see:
https://github.com/phantomics/april

In mcode the 'member of' operation is always written as:

```
x ϵ y

    0 ϵ [ 0 1 2 ]

    result is 1

    'x' ϵ 'abc'

    result is 0
```

In the discussion below vectors are synonymous with lists, and may contain varied data types.

Strings are treated similarly to vectors for read access, but in both JavaScript and in Python strings are immutable. Changes must be assigned, or temporary values can flow through function calls.
For example:

```
↑ is the 'take' primitive

mcode               result
```

```
-----                 ------
2 ↑ 'abc'             'ab'
1 ↑ 2 ↑ 'abc'         'a'
```

Expression Development

Recall that mcode expressions are evaluated right to left ←
Therefore the rightmost element must be data, and the next element
must be a function or primitive.

As more terms are chained together, data flows right to left ←

A useful primitive to see the data as it flow is □
This logs the data and passes the data through itself.
For example:

```
1 ↑ □ 2 ↑ 'abc'
```

log shows:

```
'ab'
'a'
```

'core' implementation of mcode and Data Types

Most mcode primitives process scalar numbers, strings, and vectors.
Some also process maps, tables, etc. where a useful operation may exist.

Since mcode may also compile or transpile to strongly typed languages
provision is made for 1) data type declarations, 2) extended support in the
'core' for dynamic casts and similar mechanisms for runtime type handling.

'core' implements the mcode language for a particular runtime ecosystem
by building from a 'nano' transpiler, explained below.

nb. LISP is similarly constructed by building itself from
a small set of specific operations and definitions.

There is a 'core' for JavaScript, another for Python, and others may be
developed for additional languages and targets.

Scoping and Closures

The scope of functions, objects, and data is defined by the rules of
the underlying language. In general a function definition can define
its local variables following the : as a single non-spaced token
for the underlying language. In JavaScript 'let' is inserted
before the token.

For example:

mcode

```
----------
▽ foo : x=1,y=2
    ▽ x goo y
```

output (complete)

```
------
function anonymous(_cp) {
    foo = (_a,_w,_d) =>  {
```

```
        let x=1,y=2 ;  return goo(x,y,null);
    }
}
```

nb. _cp is a 'context pointer' which is an object to hold data and code for mcode. The symbol ⊙ accesses this.

nb. x and y were declared as local and initialized

nb. in most examples we do not show the outer anonymous function that wraps the output of ≛ mexec.

≛ mexec is both the transpiler and is a primitive. However, for this discussion the outer function matters.

function 'goo' was not declared but is called with the mcode signature as op(L,R,d) or op(α,ω,δ)

'goo' will be searched using the lexical scoping rules of the underlying language. In this case a Reference error for 'goo' will be generated.

In JavaScript an outer function or object forms a 'closure', which is a function block that is searched at evaluation time. This is called 'lexical scoping' since the source code structure defines the scoping.

Both JavaScript and Python also can perform dynamic scoping (i.e. immediate data lookup or searching to resolve unknown references) but only in their console and REPL, respectively.

nb. Scoping can be clearly seen and experimented with in most debuggers.

Nested Functions

Example:

```
∇ foo : a
    a = 1
    ∇ goo
        ⍛ 22
    ⍛ 11
```

is valid, and will create a local function goo inside as foo.

Output

```
------
foo = (_a,_w,_d) =>  {
    let a ;  a = 1;
    let goo = (_a,_w,_d) => {
        return 22;
    }
    return 11;
}
```

foo() is a global function since it is at the top level
goo() is local to foo()

```
        nb. foo() can be forced to be local by writing:

        let_ foo
        ∇ foo : a
            ...

        where let... and ∇ foo are not indented


The Symbol Idea of mcode (inspired by APL)

    Symbols define standard operations.
    Symbols also defined statements that control program flow.

    Words and text are used to define data and named functions.

    Symbols are intended to be generally meaningful and recognizable.
    These are called 'primitives'

        (partial list)

        ϵ   membership
        ⌈   ceiling or maximum
        ⌊   floor or minimum

    Primitives are often polymorphic and can operate on many different datatypes,
    such as scalar values, vectors/lists.  Some primitives can also operate
    on maps and matrices.

        In APL, if unsuitable data is provided to a primitive, the 'DOMAIN' error
        was encountered.  If non-conforming data was encountered one possible
        and common error is a 'LENGTH' error.  Currently these errors are
        not implemented in mcode.

    There are also symbols for program flow.  These are 'statements'

        (partial list)

        →   'implies'  an 'if' statement (from math) A → B  as in A implies B
        ◇   'else'     resembles a flow chart decision diamond

        ∇   'function' declaration
        ⍛   'return' from function with an optional value

    mcode statements in a function execute top down ↓
    Within a statement, there are often expressions,
    and data flows from right to left ←

    Some symbols suggest their function from the shape or flow, as in:

        ←   assigns a matrix (n lines) or named function expression (1 line)
            (this is a statement, not an expression)

        ↓   push data onto a stack or 'drop' items from vector
        ↑   pop data from a stack or 'take' items from a vector
        «   insert item to head of list
        »   remove first item from a vector or list (use take)

        ⊛   logarithm, resembles a cut log
        ○   circular functions (trigonometry)
```

```
    θ   trigonometric functions (some overlap with circular functions)

    ⍒   descending sort
    ⍋   ascending sort

    ⌽   rotate string, vector or matrix, also ⊖
    ⍉   transpose a matrix

    []  bracket is used to index into vectors, to extract or set data, or
        read/write from map
```

Some symbols are common from arithmetic and calculus:

```
    +   addition, extended to vectors
    -   unary negation, minus, extended to vectors
    ×   multiplication
    ÷   division

    ⍳   iota, generate a range of numbers
    ?   generate random numbers or compute binomial distribution
    !   compute factorial
```

Some symbols are from linear algebra:

```
    +.× inner or dot product (vectors must conform in shape)

    ⌹   matrix divide / inverse,
        and least squares fit if the matrix is not square.

    ∘.  outer or cartesian product

        L ∘.F R

        function F is applied to all data L and all data R
        The result is a matrix.

    Complex and rational numbers and math may use a pair notation (TBD)
    Support for quaternions may use a vector notation
    and is also contemplated.
```

Some symbols are from logic and set theory:

```
    ∩   intersection of data (items common to sets, or unique items)
    ⊂   extract subsets from data (partition a set)
    ∪   union of data (combine sets)

    ∨   logical or
    ∧   logical and
    ~   logical not or invert
```

Symbols useful from programming:

```
    ++  increment scalar integer built-in
    --  decrement scalar integer built-in

    reflexive scalar built-in operations, including:

    += -= /= *= %=
    &= |=               bitwise mask operations
```

Constants:

```
    π   pi
    ė   Euler e
    ∞   infinity (a symbol that maps to the IEEE infinity constant, TBD)
    ϴ   zilde or nothing, maps to the null or nil value in most languages

Generally, boxed symbols deal with program flow or input/output.
Some boxed symbols are 'statements' that have a specific syntax.

    The ∇ symbol declares a named function.
    This is also legacy APL, and is not 'boxy' in appearance.

'boxy' shaped program flow statements are:

    ⍛   return from function    'boxy' exit of ∇ function declaration
    ⍥   for statement (iterate) wheel in box to represent processing
    ⍈   while statement         equality test in box
    ⍗   break                   break the program flow downward
    ⍐   continue                continue the program flow upward,
                                to the top of the loop
    ⍁   switch / case           'boxy' decision diamond

    Legacy APL defined ⌹ which is matrix divide.  This is 'boxy' and pertains
    to matrices, not functions.

    Also, → if and ⋄ else are not 'boxy'

Primitive symbols are mostly 1 character in length.

    An exception is ?= for the assert primitive (defined in file 'core')

Most symbols are 'primitives' that operate on single values (scalars)
or vectors (lists)

    ⍴   length or size of vector or string data

A few symbols take a function or primitive and apply it to data,
for example:

    ¨   each    apply a function to each item of data, returns a result vector
    ⌿   reduce  apply a function on each item in ⍵  with an accumulator ⍺
                and return a scalar
    ⍀   scan    apply a function between each pair of data items and
                returns a vector

Some symbols perform operations useful with string data:

    ≡       regular expression test or search test or startsWith test
    ≠       regular expression replacement
    ⍋       to lowercase (TBD)
    ⍒       to uppercase or sentence case (TBD)
    ⍦       split string into tokens (TBD)
    ⍪       concatenate or join strings (or + for JavaScript built-in)

    ⍙.j     change data to JSON or parse JSON
            (also create map and Date/Time objects, calls 'new')

    ⍢       delete data (calls 'delete')

    ⍕       format numeric data (can use sprintf C-style format specifier)
```

```
    +       concatenate strings immediately (in JavaScript)

    +=      concatenate with assigment for strings,
            and increment scalar numeric (built-in)

Symbols useful for web development: (TBD)

    ∘   document object model (DOM) operations


Data Literals

    0 or 0.0    numbers in base ten
                numbers in other bases or complex numbers may
                be supported by the underlying language

    'abc'       string
    "abc"       string
    `abc`       multiline string
    /abc/       regular expression literal

    {}          object literal example: {k1:'v1',k2:2}  note -no spaces-
    []          vector literal example:  [ 1 2 3 ]      note no commas
                                                        -has spaces-

    abc         variable
    a.b         object or structure a with property or field b

    foo         function foo if used in op position as in 'L op R'
    foo()       function foo called in pre-fix notation
    foo(a,b)    called with args a,b etc - usual C style calling
                note no spaces to cause mcode to pass call to
                underlying language


Create primitive

    Modifiers produce the following:

    modifiers                   output
    ---------                   ------
    Δ.v m,r                     let m,r;
    Δ.n Class                   new Class();
    [1,2,3] Δ.n Class           new Class(...[1,2,3]);


    The create primitive creates complex data objects:

        console input marked with >

    Δ @                         date/time, current time

        > Δ @
        D 7/26/2024, 10:13:09 AM        // now in local time
        > ∓.j Δ @
        "2024-07-26T17:13:25.342Z"      // now in UTC (Zulu) time

        > ∓.j "2024-07-26" Δ @          // set date and check
        "2024-07-26T00:00:00.000Z"      // ISO format

    Δ {}                        empty map
```

```
> P = [['a',0],['b',1]] ⍙ {}     // key-value pairs in array to set map
M {  a 0  b 1  }
> 'c' [].2 P                     // set entry
M {  a 0  b 1  c 2  }
> 'c' [] P                       // get entry
2
```

`[ n m ] ⍙.i [[]]` n x m matrix, .i sets identity, otherwise zeroes

```
> [ 4 4 ] ⍙.i [[]]
[  1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   0, 0, 0, 1  ]
> [ 2 3 ] ⍙ [[]]
[  0, 0,
   0, 0,
   0, 0  ]
```

Tables (NYI)

Several primitives perform special handling to support tables.
Tables are similar to spreadsheets and databases in certain ways.
Tables can be serialized to be readable in plain text also.

⌹	invert table to a set of column vectors
⍙	gather column vectors and create a table object
⍤	format a table for text dump or HTML presentation

About Fonts and UTF-8

Symbols on this page and in mcode, use Unicode and will usually
display correctly.

This does not work well if a web server does not inform the browser that
the contents are UTF-8, and similarly some text editors may not handle
UTF-8.

It is recommended to use the APL386 font which will render most symbols in
a single character. Other fonts will display, however columns may not
line up, and some characters will be constructed from multiple glyphs
(this causes problems in editing text unless APL386 is used.)
Another useful font is the older APL385 font.

The APL386 and APL385 fonts were developed for the APL language,
such as GNU APL and Dyalog APL.

Grouping Symbols

( )	groups expressions for evaluation
[ ]	groups items that belong to a vector or matrix
{ }	groups expressions will define an anonymous function (lambda or deferred function)

indenting groups statements into blocks for
named functions ∇ and most statements

```
    Note that JSON also uses {} [] ,
    These symbols will be ignored by mcode if they are not surrounded by spaces.
    This allows embedding of JSON inside of mcode.


Primitives from the 'nano' transpiler:

    In order to define the 'core' which builds the language, a few essential
    primitives are provided by a 'nano' transpiler:

    □   writes to stdout, a log, or an HTML textarea
    ⍎   translate and execute mcode
    ⍁   passes a statement to the underlying language
    ⍙   adds a primitive, available at the next execute ⍎ invocation

    These are explained further below.

    It is also contemplated to write the 'nano' transpiler in mcode, which then
    may make porting to new lanugage platforms easier and clearer.


Primitives defined in 'core'

    ⍬   theta       trig functions
    ⍷   typeof      returns the type of data
    ⍙   create      create complex data types such as maps, date/time
    ↓   push
    ⍞   system
    ∈   memberof
    ⍴   length
    ⍳   range
    ¨   each
    ⌿   reduce
    ⌷   brackets
    ⍁   read        read file or load a URL
    ⍂   write       write file to server

    These are defined in a mix of mcode and javascript.
    As new languages become supported 'core' will define these primitives
    also in their respective underlying language.


Built-in ops:

    ops from the underlying language are frequently not vectorized, and also
    may have an operator precedence specified in the underlying language.

    built-in ops use the underlying language
    (not currently vectorized for JS, Python):

    =   assignment
    ≈   equality test with type conversion  or ==
    ≒   exact equality test (no conversion)  or ===
    ~   not  or !
    ≤   less than or equal to  or <=
    ≥   greater than or equal to  or >=
    ∨   logical or  ||
    ∧   logical and  &&


Mixing mcode and Underlying language:
```

In general, code with spacing is treated as mcode and code that
has no spaces is passed to the underlying language.

Parts of expressions that are not mcode can be mixed in by omitting spaces.
For example:

```
    input       output
    -----       ------
    v?x:y       v?x:y           // ? is the ternery operator in JS
                                //   returns x if true or y if false
    a ? b       ?(a,b)          // ? is the 'roll' or 'deal' primitive
                                //   returns random numbers
```

As mentoned above, the symbols that are converted to functions and
part of mcode are called 'primitives'

Primitives are translated to runtime library function in the
_ object (for JavaScript)

```
?(a,b)  thus becomes  mcode.deal(a,b)
```

Overloaded Primitives and Built-ins

Primitives and Built-ins are both infix notation, like 0 + 1

```
mcode                       output
-----                       ------
0 + 1   // built-in +       0 + 1;
0 +. 1  // vectorized +     mcode.plus(0,1,null);
0 +.× 1 // inner product    mcode.plus(0,1,'×');
0 ∘.+ 1 // outer product    mcode.outer(0,1,'+');
```

nb. + was added as a primitive, and is -also- a built-in
(in the 'bmap' table in the transpiler)

nb. ∘ is 'outer' This primitive performs a cartesian product
of function δ over each α and each ω

nb. 0 and 1 are stand-in args, actual args would be vectors or matrices

Operators

In most computer languages an 'operator' is an operation such as + - % (modulus)

In mcode, these symbols are either built-ins (provided by the underlying language),
or they are vectorized functions.

The term 'operator' in mcode (as in APL) refers to a function that operates
on another function.

The main operators (from above) are:

```
    ¨   each    apply a function to each item of data, returns a result vector
    ⌿   reduce  apply a function on each item in ω  with an accumulator α
                and return a scalar
    ⍀   scan    apply a function between each pair of data items and
                returns a vector
```

To iterate of over a list x, one can write statements using a loop:
where b is local var, and foo(α,ω) is a function:

mcode	output
b ⍤ x : foo b	for (b of x) foo(null,b,null);

However, another way is to use an 'operator' with a lambda function:

¨ each applies a function α to each element in ω

We can write:

mcode	output
{ foo ω } ¨ x	mcode.each((_a,_w,_d) => (foo(null,_w,null)),x,null);

nb. the lambda is implemented in JavaScript as an arrow function =>

If there is an argument in the L position this will be moved to δ (modifier)
for the operator function. For example:

mcode	output
y { foo ω } ¨ x	mcode.each((_a,_w,_d) => (foo(null,_w,null)),x,y);

nb. This grouping is still L op R where 'op' is the lambda function

TBD: operators on primitives, such as:

```
y + ¨ x
```

currently this must be written as:

```
y { α + ω } ¨ x
```

Notebooks

The Advanced Code Editor (ACE) is used to provide both editing and a session log.
This is an open source web based editor with syntax highlighting, code folding,
and a syntax checker for JavaScript. It is popular on many web sites.
More information is here: https://ace.c9.io/

A capability, called "notebooks" was added to incorporate
images, SVG graphics, animated GIFs, LaTex equations,
general HTML into the text editor.

The general syntax is:

```
/// notebook identifier lines type data
```

where:
- identifier is a unique name used as an id for the HTML element
- lines is the number of text lines to reserve
- type is one of: img, tex, html
- data is:
  - for img: a URL or filename of an image, SVG, or GIF
  - for tex: a LaTex expression rendered by MathJax
    see https://www.overleaf.com/learn for an overview of Tex for equations

```
        for html:   a 1 line HTML string to be inserted into the document

    /// will be seen as a comment, or can be prefixed with # for Python

    Examples:

    /// notebook nb1 5 img ace/kitten.png

    /// notebook nb2 10 img ace/plot1.svg
    /// notebook nb3 4 tex //int x^n dx = //frac{1}{n+1}x^{n+1}, //hspace{1ex} n//neq-1
    /// notebook nb4 2 html <button onclick="alert('ok!')">ok</button>

    notes:
        * nb. for tex, // are replaced to space\ due to JavaScript handling of escapes

        * MathJax must be loaded by page if 'tex' is used, eg:
        <script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
        <script id="MathJax-script" src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-
chtml.js"></script>


Object Oriented Programming

    Classes can be declared and instantiated using ∇ and ⍙
    or the ⍁ pass-through statement for more complex cases.

    Useful statments are:

        ∇ class                     // base class
        ∇ class : base              // class inheriting from base
            ∇ method                // method  nb. is indented

        ∇.s varList                 // simple constructor, sets 'this'
        ∇.c varlist                 // general constructor function
            ∇.t varlist             // set 'this'

        var = arglist ⍙.n class     // instantiate class
            or
        new_class(arglist)

        ⍁   pass-through

        ⊙   the mcode.cp shared object
        ⍙   this

        in mcode for JavaScript, super() is not a symbol, and can be written out


    Example:

        ∇ OWL               // base class
            ∇.s x,y         // simple constructor
            a = 0           // static
            ∇ m1            // method
                ⎕ 'm1'
                ⍵ ω
            ∇ init
                'init ⍙' ⎕ ⍙
        ∇ FS : OWL          // derived class
            ∇.c fs,x,y      // general constructor
                super(x,y)  // call superclass
```

```
            ∇.t fs      // init 'this' from list
            □ 'cFS'
        ∇ m1a           // method
            □ 'm1a'
            □ Δ.fs
            ⍛ ω
    s1 = [ 1 2 3 ] Δ.n FS
    □ s1.m1 11
    □ s1.m1a 22
    □ s1


output
------

    class OWL { // base class
        constructor(x,y) { this.x=x??0; this.y=y??0;  } // simple constructor
        a = 0; // static
        m1 = (_a,_w,_d) => /* E030 fix */ { // method
            mcode.quad(null,'m1');
            return _w;
        }
        init = (_a,_w,_d) => /* E030 fix */ {
            mcode.quad('init Δ',this);
        }
    }
    class FS extends OWL { // derived class
        constructor(fs,x,y) { // general constructor
            super(x,y); // call superclass
            this.fs=fs??0;  // init 'this' from list
            mcode.quad(null,'cFS');
        }
        m1a = (_a,_w,_d) => /* E030 fix */ { // method
            mcode.quad(null,'m1a');
            mcode.quad(null,this.fs);
            return _w;
        }
    }
    s1 = new FS(...[ 1, 2, 3 ]);
    mcode.quad(null,(s1.m1(null,11,null)));
    mcode.quad(null,(s1.m1a(null,22,null)));
    mcode.quad('s1',s1);

runtime
-------

    cFS
    m1
    11
    m1a
    1
    22
    s1 = {...} // object dump

nb. the /* E030 fix */ comment above is needed due to a bug in
    jshints code checker, as of version 2.13.6

nb. advanced:

    In the example below, either method call will retain 'this' pointer:
```

```
    setTimeout(()=>s1.m1(3,4),1000)
    setTimeout(s1.m1,1000)

    see https://javascript.info/class for a discussion of "losing this"


Example, using pass-through:

mcode
-----
⊙.Foo = class
    a = 1

⍁ ⊙.f = new ⊙.Foo

output
------
_cp.Foo = class {
    a = 1;
}
_cp.f = new _cp.Foo


Example, with constructor and using ⍛ macros:

    nb. the prefix ⊙ places the class definition and instances
    in the shared mcode.cp object.  This is useful to see
    these objects in the debugger, since they will persist.

    If ⊙ is omitted then Foo, f, g will be deleted when the
    mexec temporary function completes.

    ⊙ could be a library, module, class to add class Foo
    to a larger existing object.

mcode
-----
⍁ ⊙.Foo = class
    a = 'ok'                // a is common to all instances
    ⍛.s x,y,z               // x,y,z are initialized per instance
    ⍛ m                     // method m
        ⍞ 'm'
        ⍗ ω

⊙.f = [ 1 2 3 ] ⍛.n ⊙.Foo   // new instance using ⍛ macro
⊙.g = ⍛.n ⊙.Foo
⊙.g.m 1                     // call method 'm' on instance 'g'
⍞ ⊙.f                       // show contents of instance 'f'
⍞ ⊙.g                       // show contents of instance 'g'

output
------

runtime
-------


Notes

    The use of ⍁ pass-through allows more complicated language-specific
    syntax to be used, such as 'class Foo extends Goo' etc.
```

```
    The use of ∇ macros has the advantage that the code may be more
    portable between languages.  eg. JavaScript uses 'new' while Python
    does not use 'new'


Asynchronous Programming

    mcode core provides the ⍇ read primitive to read files as data, or to load
    entities (i.e. HTML, CSS, scripts) into the current web page.  Such read
    operations do not block the caller, and instead are carried out asynchronously.

    In many cases, however, it is desired to perform steps in a sequence,
    which complicates program logic.

    JavaScript uses an asynchronous architecture for many operations,
    either events with callback functions, or Promises.

        nb. The ⍇ read primitive should be universal in all language versions of
        mcode.  However, asynchronous programming, and 'await' in particular
        is specific to JavaScript.

        nb. ⍏ mexec also creates a Promise called mcode.busy which can be
        used to pause the IDE until I/O or async operations complete.


    In-depth:

    '.then' callbacks on Promises

    A different style of programming can be mixed using hybrid style:

    F(x,y).G(z)

        In this kind of expression F is called with arguments x and y
        F returns an object with method G, called with argument z

    A special method used in asynchronous programming: '.then'
    mcode hybrid expressions can be written such as:

    F(x,y).then(G(z))

        note that these kinds of expressions read and execute
        left to right → per the rules of the underlying language.


    Using 'await' instead of '.then'

    ∇.a foo         declares the function 'foo' to be asynchronous
                    In the JavaScript ecosystem this means that the
                    function can then use 'await' for a Promise.

    About async/await and Promises in JavaScript:

        async functions:

        - start normally, but when an 'await' is encountered the remainder of the
          function is scheduled -after- the main stack completes
        - so, make Promises -before- calling async fns

        - they run until an await clause
```

```
        - do not block the caller, since they are added to the "microtask" queue

        - functions should call resolve to release any awaits on the Promise
        - otherwise any awaits will hang indefinitely
        - or reject to throw error (should use try/catch in the async)

        - return a Promise, which is created implicitly if a Promise is not returned
        - the caller can also be async and await the async fn for its data result

        - fetch uses 2 Promises and those call resolve/reject that can trigger .thens

    See the examples and tests in 'core'


Special Symbols

    Δ   symbol for 'this' or 'self' in Python

        zvzv TBD: if used in the 'op' position then calls current function recursively

    ⊙   symbol for shared execution context
        allows passing data between various calls of ⍎ execute
        resolves to the mcode.cp object

    .   selects elements or fields from an object as in
        object.field


Resolving Symbols with Dual Meaning

    Some symbols have dual meaning, such as ? and !

    In mcode and APL ! is factorial (with no Left argument, called 'monadic'),
    or if called as L ! R it is binomial (and called 'dyadic' in
    APL terminology).

    In many languages ! means 'not'.
    The symbol ~ can be used to mean 'not' in mcode phrases.
    Example:

    input               output
    -----               ------
    !a                  !a
    ~ a                 !a              // replacement of ~ symbol to ! operator
    ! a                 !(null,a)       // L not given, compute factorial of a
    a ! b               !(a,b)          // compute binomial or beta fn of a,b


Parenthesis in mcode Expressions

    Parenthesis may be needed to group data properly with primitives.
    Evaluation is strictly right to left, and there is no operator precedence.

    Example: + is not higher in priority than ∈ so ( ) are needed:

    ( a.b + 1 ) ∈ [ 1 2 ]           mcode.memberof(( a.b + 1 ),[ 1, 2 ])

    a.b + 1 ∈ [ 1 2 ]               a.b + ( mcode.memberof(1,[ 1, 2 ]) )

    nb. structured data access is supported, in this example 'a' is an object,
    and 'b' is a member
```

nb. + has -not- been overloaded as a primitive at this time, so the built-in addition operator only operates on scalar values.

The Left argument to a primitive can be omitted.

```
1 ϵ ι data      // ι has no Left argument
1 ϵ ( ι data )  // ι has no Left argument

2 - 1           // 2 minus positive 1
2 + ( - 1 )     // 2 plus unary negation of 1
```

nb. at this time + and - are scalar built-ins

When primitves are mixed with built-ins, it is recommended to use ( ) to specify the order in which values should be computed.

Defining New Primitives

mcode is dynamic and new associations from symbols to functions can be made at runtime. Most symbols are defined in file 'core' which focuses on general purpose computing.

The default runtime library object for mcode is named 'mcode'

Other libraries or Classes can be added to 'mcode', and new primitives connected using the ⍙ create primitive or mcode.addPrim() function.

An additional file such as 'numerics.mc' could be written to define vectorized numeric operations, and similarly for symbolic math.

In the Python version of mcode numerics would likely map to the 'NumPy' and 'SciPy' packages.

Example:

To define ⊥ as a primitive that could handle vector data:

in the 'nano' transpiler:

remove any line from table of built-ins 'bmapIn' that might use that symbol.

```
'         +       x           // addition        reflexive'
```

define a mcode.plus primitive function (in mcode or in the underlying language)

add the primitive to the language:

```
'⊥' ⍙ 'mcode.newFunction'
```

or

```
'mcode.addPrim( '+', 'mcode.newFunction' );'
```

nb. Julia language does allow functions to be directly named in Unicode symbols without transpiling, however

1) the L op R triplet concept and
2) right to left evaluation are not part of Julia.
3) Also statements are traditional reserved words rather than symbols.

```
    Useful Web Resources for Symbols

        see:
        https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=\p{subhead=APL}
        http://xahlee.info/comp/unicode_APL_symbols.html
        http://xahlee.info/comp/unicode_support_ruby_python_elisp.html
        http://xahlee.info/comp/APL_symbol_meaning.html


Scalar Equality Conventions

    In many C based languages, = means assignment, == means equality test.

    mcode uses the following convention:

    input               output
    -----               ------
    a==b                a==b    // scalar equality test
    a ≈ b               ≈(a,b)  // equality test primitive
                                // works on most datatypes, eg. vectors
                                // nb. ≈ is currently a scalar built-in
    a ≒ b               a===b   // a is identical to b (objects at same address)
    a=b                 a=b     // assignment
    a = b               a = b   // assignment

    As noted above:

        Primitives can be defined for + - × ÷ % and = to operate over vectors.

        + also means string concatenation in JavaScript or addition

    nb. At this time ≈ ≠ ≒ have -not- been extended for vectors
    = is assignment and will work for any datatype


Passing Keywords to the Underlying Language

    The symbol _ can be appended to a word to pass that word
    to the underlying language.

    input               output
    -----               ------
    let_ x=0            let x=0

    Syntax coloring in the editor provides some guidance.

    Each invocation of ± removes one level of _
    This is similar to escape handling of \ in strings.

        'core' has some functions such as ∇ delete that
        have several _ due to nested ± levels

    Similarly, strings must be escaped:

        backquotes ` inside of backquotes ` can be escaped as \`

    ⍞ ...

        passes the following string unaltered into the underlying language

        nb. this is useful for more advanced object oriented programming,
```

```
    or syntaxes in the underlying language that are not possible or
    simple to write in mcode, such as OOP with inheritance.

⍝.err
    emits /* jshint ignore:start */
⍝.nerr
    emits /* jshint ignore:end */

    nb. these are needed for a case where jshints parses class syntax
    incorrectly, and perhaps for other cases
    (as of jshint version 2.13.6 in 4/2024)
```

Statements Syntax

Above we introduced the ∇ function statement and used single expression statements. Example:

input

```
⊙.C={}              // an object in JavaScript
∇ C.add
    ⍛ α + ω         // scalar addition of L + R arguments
⍞ 1 C.add 2
```

output

```
function anonymous(_cp) {
    _cp.C={};
    C.add = (_a,_w,_d) => {
        return _a + _w; // scalar addition of L + R arguments
    }
    mcode.quad(null,( C.add(1,2) ))
}
```

result

```
3
```

A few more statements are introduced.

Note that the If statement uses in-fix notation as A -implies- B

```
If statement:       code blocks are indented as in Python
    A → B           // if e1 then e2
    ⋄               // else

    any statement or block may follow 'else'

Assignment:
    x = e1          // assignment to variable x
                    // the scope of x depends on where it was declared
                    // nb. this is called lexical scoping

Comments:
    // cmt        input right of // is a comment
    # cmt        python style comment
    ⍝ cmt        APL style comment using lighthouse ⍝

    /* inline or multiline comment */
```

```
        // -err         disables error reporting from JShint for that line

    Strings:
        single or double quote
        multiline strings use backquote `abc` or """ as in Python


Vector and matrix declarations

    ← is a statement that assigns a vector or matrix and can span several lines

    input                   output
    -----                   ------
    V ← [ 1, 2, 3, 4 ]      V = [1,2,3,4]
    M ← [ 1, 2,             M = [[1,2],[3,4]]
          3, 4 ]

    mcode is parsed in each data element, and then the
    data elements are evaluated by the underlying language
    -comma space- separates the elements

    input
    -----
    Mz ← [      cosθ t,     - sinθ t,       0,  // rotate on Z axis
                sinθ t,     cosθ t,         0,
                0,           0,             1   ]

    output
    ------
    Mz =  [[mcode.vf( null,t,[Math.cos,null] ),- (mcode.vf( null,t,[Math.sin,null]
)),0,],[mcode.vf( null,t,[Math.sin,null] ),mcode.vf( null,t,[Math.cos,null] ),0],[0,0,1]];

        nb. mcode.vf is a runtime function that applies functions over vectors
        and matrices, and it will be discussed below.

        nb. Comments in the matrix are allowed and are stripped out.

    runtime
    -------
    Mz = [ [ 0.00000 -1 0 ] [ 1 0.00000 0 ] [ 0 0 1 ] ]


    Another example:

    input                   output
    -----                   ------
    sinθ 2*π*t              mcode.vf( null,2*Math.PI*t,[Math.sin,null] );

        nb. since 2*π*t is not spaced, it is treated as a scalar expression,
        and is processed by the underlying language, not mcode.

    sinθ [ 0 1 2 ] × t      mcode.vf( null,(mcode.vf( [ 0, 1, 2 ],t,[(a,w)=>a*w,null]
)),[Math.sin,null] );

        nb. since the expression is spaced, this is treated as a vector function,
        and transpiled by mcode.


Vectors in expressions

    Vectors can also be part of expressions, and the elements do not need
```

comma separators.

mcode is NOT parsed in these vector expressions, since the vector terms do not follow the L op R triad format, but data and functions can be called in the underlying language.

```
input                          output
-----                          ------

V = [ 1 2 3 4 ]                V = [ 1,2,3,4 ]
V = [ a b c() ]                V = [ a, b, c() ]

□ [ 1 2 3 4 ]                  mcode.quad(null,[ 1, 2, 3, 4 ])

'V is' □ [ a b c() ]           mcode.quad('V is',[ a, b, c() ])
```

mcode.quad is a built-in function that writes to stdout, a log, or an HTML textarea

Using Primitives provided by the transpiler

Essential primitives are provided to build up the language dynamically.
The mcode core runtime is defined in the 'core' file.
For performance reasons, this file is usually cached as pre-transpiled in the underlying language, eg. 'core.js'

The essential primitives are:

□ write message to stdout, a log, or an HTML textarea

```
input                          output on log
-----                          ------
□ 'message'                    message
'a' □ a                        a = value
```

⍎ translate and execute code (mix of mcode and underlying language)

ω is the input as a string

α is an optional named error handler function called as
α(errorEvent, outputCode)

If there is no error handler provided in α,
then an exception of 'error' is thrown on any kind of caught error at eval time

δ flags: any of 'medr' default is 'e'
m write input to log e execute
r return code output as a string
d show lexer, expression builder, and parser internals
nb. can be a lot

The output of mexec is an anonymous function that is immediately called

⊙ is a symbol that refers to a context object that may be used for dynamic variables.

For example:

⊙.M ← [1, 0, 0,

```
                0, 1, 0,
                0, 0, 1, ]
        ⊙.N ← [ 0, 1, 2 ]

        '[[]]' ?= '∈ ⊙.M'    // assert that M is a matrix
        '[]' ?= '∈ ⊙.N'      // assert that N is a vector
```

nb. For JavaScript, data declared in the ⊙ context resides in the mcode.mexec or ≛ closure, and will live as long as any function has a reference to that data.
These data are not global; in the above example M and N are freed when the ≛ that ran the above code completes -and- no other functions holds a reference to M or N.

If the input is a single statement then 'return' is prefixed
otherwise use ⍛ the return statement to return a result

Since mexec is computationally costly, the 'r' option
may be used to obtain the transpiled code, and, for example save
it to a .js script file.

At this time such caching is not automated, as it is in Python for example.

⍰ is a transpiler directive and control primitive

Other operations of ⍰ show the transpiler tables

∆ add a new primitive to mcode

'⌈' ⍁ 'library.max'

this registers the symbol ⌈ with the function in library.max
where library is a class or object containing functions

nb. adding a new primitive is not recognized until a subsequent use of ≛

I/O

⍇ read files

⍈ write files

In the JavaScript ecosystem, the ⍇ read primitive uses 'fetch' to read a file and attach it to the current document.
For example,

```
{ foo 0 } ⍇ 'https://website/file.html'
```

function 'foo' will be called after the file is loaded by the browser.
Currently, .html and .css file types are recognized.

See file 'core' for more info.

Modifiers

Primitives, such as ≛ above can have modifiers, syntax uses a . with no space.

For example,

```
    input                          output
    -----                          ------
    ⍎.er '1 + 1'                   r = mcode.mexec(null,'1 + 1','er')

    All mcode runtime functions are called as mcode.fn(L,R,modifier)

    Where 'fn' was translated from the known symbols added previously using ⍙

    Within mcode functions, these arguments can be symbolically referred to as:

    ⍺   for Left argument (could contain null if not given when called)
    ⍵   for Right argument
    ⍶   for Modifier (contains null if not given when called)

        ⍪.r         catenate by rows in mcode


        APL supports multidimensional arrays.
        mcode would rely on NumPy for array support in Python, so this is TBD.

        Proposal: supporting a [ ]  modifier syntax would make mcode
        more similar to APL's axis operator, for example:

            ⍪[1]       catenate on first axis (rows) in APL (TBD)


Anonymous or lambda functions

    { ⍺ + ⍵ }           lambda or anonymous function
                        evaluation of the expression is deferred
                        useful for callbacks from gui or i/o functions, as in


    Example:
    { foo ⍵ } ⍁ 'https://website.com/filename.js'

            where ⍁ fetches data from web, local storage, or disk
            this reads the file and then attaches the URL to the current HTML document
            after it is loaded, function foo(⍵) is called
            ⍵ will contain the data that ⍁ obtained

    f ← { ⍺ + ⍵ }       named function expression (single line only)

    In expression groups, these forms are valid for lambdas:

    { ⍵ + 2 } 1             // ⍵ gets 1
    1 { ⍺ + ⍵ } 2           // ⍺ gets 1 and ⍵ gets 2

    ⍶   is a modifier as in ⍺ op.⍶ ⍵     gives op(⍺,⍵,⍶)


Parser Directive

        /// stop   causes an immediate stop of input processing

        This is useful, for example, to stop processing of unfinished code,
        or limit processing to only consider a test at the top of a file.

            nb. // stop is treated as a comment and has no effect
```

Statement End and Semicolons

mcode does not automatically write semicolon ; to the output code unless the line is a 'solo' expression (it has no other token or statement). To suppress this, write a trailing _

For example, ; is not wanted before a '.then' clause since it is actually part of a single expression. The alternative is to write the entire expression on a single line.

JavaScript in many cases tolerates missing semicolons, however in some cases problems can arise.

eg. Adding the ; solves a case where for certain lambda calls the preceding statement should be ended with ;

input	output
X ; 1 { 2 } 3	((_a,_w,_d) => 2)(1,3,null)

statement X may need an ending ;

JavaScript notes

JavaScript has some unusual traits compared to Python or C++

In general, the value undefined if encountered means that there could be an error in the code.

The value null is used to mean no value or nothing.

Since JavaScript fails silently in web pages, it is useful to use try { } catch { } blocks to locate and handle runtime errors. The mcode transpiler runs output code through 'jshint', a syntax checker, prior to JavaScript evaluation.

Execution Control: Looping

Although loops can be written explicitly using the patterns below, in the style of 'functional programming' the practice is to use operators and functions, such as ¨ each, ⌿ reduce, ⍣ power that apply a function to each element of data, without a visible loop in the input code.

The following statements are provided in order to write loops in code.

symbol	output template
⍥	for (L of R) S
⍣	while (L) S
⍗	break
⍐	continue

Statement symbols must be spaced (just like primitives), and require Left and Right arguments, which are expressions in either mcode or the underlying langage.

```
    A : can be used to separate 'S' predicate.

    For example:

    input               output
    -----               ------
    b ⍰ α : r += b      for (b of _a)   r += b

    A indented block of code can follow the 'for' and 'while' statements
    instead of 'S' above.

    space ; space may be used to separate statements


Execution Control with Switch / Case

    nb. this is not used much so far in the JavaScript version.

    The ⍰ symbol provides a switch statement construct.

    The first occurrence must use a .s modifier.  Subsequent statements
    shoudl be indented.

    Each case is written ⍰ with no modifiers, and uses : to separate the test.

    The .d modifier generates a default statement.

    Note the placement of ⍗ break for each statement in this example:

        ⍰.s α
            ⍰ 'sin'    : r = Math.sin(ω)    ⍗
            ⍰ 'cos'    : r = Math.cos(ω)    ⍗
            ⍰ 'tan'    : r = Math.tan(ω)    ⍗
            ⍰.d        : r = Math.PI


About the 'nano' Transpiler

    The transpiler is a table driven recursive descent parser.
    These tables are maps used to translate input symbols to output code.

            name    description                 purpose
            ----    -----------                 -------
            'smap'  mcode statements            statement output patterns
            'vmap'  mcode variables/constants   output phrase or value
            'bmap'  built-in functions          list of operators or functions
            'pmap'  primitives                  symbol map to function
            'fmap'  vectorized functions        small fns for each array element

        nb. these are Map data types in the transpiler

        All primitive implementations reside in the 'mcode.' runtime object.

        invocation of ⍋ (in mcode) or called as mcode.mexec() follows these steps:

step                            function        note
----                            --------        ----
process input                   'mexec'         setup to transpile program input
read program input              'parsePgm'      expression or function as string
lexical analysis of statements  'lexer'         breaks up statements into a list
```

```
build each statement             'buildStmt'     finds expressions in statements
lexical analysis of expressions  'lexer'         tokenizes by primitives
build expression tree            'exTree'        makes a tree of 'L op R' triplets
emits code from tree nodes       'emit'          uses depth-first-visit of tree
build code blocks from indents   'buildBlocks'   recreate indenting
check for malformed code         'jshint'        uses the JShint package
evaluation by javascript         'mexec'         uses 'new Function(...)'
call generated code              'mexec'         argument is 'context object'
catch parse or execution errors  'mexec'         call optional error handler
```

mcode Guide

This is from the Guide IDE button, or from mcode.showMCstate() or ⍰ 0 in the console:

```
mcode nano-transpiler version 0.06.19.2024 EXPERIMENTAL
mcode core version 0.06.19.2024
```

mcode guide, from JavaScript transpiler tables

Statements

symbol	JavaScript	comment
∇	R = (α,ω,δ) =>	// ∇ functionname : localVarList eg. ∇ f : a,b,c
		// ∇.a for async fn
⍛	return R	// nb. top level has a return value
→	if (L) S	// L implies S S is expr or stmt or block
◇	else	// must follow → if
⍣	for (L of R) S	// var ⍣ expr : expr or stmt or block
:	: S	// L ⍣ R : S S is stmt or block
⍢	while (L) S	// expr ⍢ expr S is stmt or block
⍤	break ;	// only valid in loop
⍠	continue ;	// only valid in loop
⍁	case R :	// ⍁.s switch R ⍁.d default R
⍀	R	// pass R to underlying language (useful for OOP etc)
⍝	//	// single line comment
←	L = S	// matrix and 1 line function expression assignment
;	;	// stmt separator
⍦	class R	// ⍦ class : baseClass
		// ⍦ name method
		// ⍦.s varList simple constructor method
		// ⍦.c varlist general constructor method
		// ⍦.t varlist initialization
		// ⍦.m class mixin class
⍫	delete R ;	// valid only for object.R in JavaScript, not vars

L is left, R is right, S is statement after : pr next { block }
use ← to assign a matrix containing code expressions, eg.

```
M ← [   1, t0 × π,
        t1 × π, 1 ]
```

use ← to assign a named 1-line function expression, eg.

```
avg ← { ( 0 + ≠ ⍵ ) ÷ ⍴ ⍵ }
```

Primitive Functions

```
⍰ mcode.output
⊢ mcode.nyi
⍎ mcode.mexec
```

```
               ⍙ mcode.create
               ∈ mcode.typeof
               ⍞ mcode.system
               ⍴ mcode.shape
               ↓ mcode.push
               ⍪ mcode.concatenate
               ⍳ mcode.iota
               ⌷ mcode.brackets
               ¨ mcode.each
               ⌿ mcode.reduce
               ⍣ mcode.power
               ?= mcode.assert
               ↑ mcode.pop
               « mcode.insert
               » mcode.remove
               ≡ mcode.match
               ≠ mcode.replace
               ⊃ mcode.split
               ⊂ mcode.join
               ∈ mcode.memberof
               ⍁ mcode.read
               ⍂ mcode.write
               ⍜ mcode.dom
               ○ mcode.quat

      primitives and functions are called as  α fn ω
      expressions execute right to left as    α fn2 fn1 ω  same as  α fn2 ( fn1 ω )
      fn can have a modifier as               α fn.mod ω

      α ⌷ ω  is useful to trace function execution, eg.
      avg ← { ⌷ ( 'step 1' ⌷ 0 + ⌿ ω ) ÷ 'step 2' ⌷ ⍴ ω } ⍝ avg [ 1 2 3 ]
          step 1 = 6
          step 2 = [ 3 ]
          [ 2 ]

      ⍞       alone sets a breakpoint for the JavaScript debugger


   Variables and Constants
   symbol  JavaScript              comment
   ------  ----------              -------
   α       _a                      // left function argument
   ω       _w                      // right function argument
   ∂       _d                      // modifier or 3rd function argument
   ⍬       null                    // null value
   π       Math.PI                 // pi
   ė       Math.E                  // Euler e
   ⊙       _cp                     // shared object  same as  mcode.cp
   ∆       this                    // this  for object oriented programming


   Vectorized Functions
   fn      fn ω                    α fn ω
   --      ----                    ------
   ≈       mcode.nyi               (a,w)=>a==w             // no-op, equality test
   ≠       mcode.nyi               (a,w)=>a!=w             // no-op, inequality test
   ⍎       (a,w)=>Math.trunc(w)    (a,w)=>+w.toFixed(a)    // truncate, special rounding
   ~       (a,w)=>+!w              mcode.rmset             // logical not, remove from
set
   +       (a,w)=>+w               (a,w)=>(+a)+(+w)        // use +.  cvt, add  α +. ω
   -       (a,w)=>-w               (a,w)=>a-w              // negate, subtract  α -. ω
```

```
    ×       (a,w)=>Math.sign(w)     (a,w)=>a*w              // sign of number, multiply
    ÷       (a,w)=>1.0/w            (a,w)=>a/w              // reciprocal, divide
    /       (a,w)=>1.0/w            (a,w)=>a/w              // reciprocal, divide
    |       (a,w)=>Math.abs(w)      (a,w)=>a%w              // absolute value, α modulus ω
    ⍦       (a,w)=>Math.round(w)    'norm'                  // round, norm of vec ω using
hypot
    ⌈       (a,w)=>Math.ceil(w)     (a,w)=>Math.max(a,w)
    ⌊       (a,w)=>Math.floor(w)    (a,w)=>Math.min(a,w)
    ⊗       (a,w)=>Math.log(w)      (a,w)=>Math.log(w)/Math.log(a)  // log(base=α,x=ω) = y
    ?       (a,w)=>Math.random(w)   (a,w)=>Math.round(a*Math.random()) // uniform dist
    ⍲       (a,w)=>Math.sqrt(w)     (a,w)=>Math.pow(w,1.0/a) // α ⍲ ω
    **      (a,w)=>Math.exp(w)      Math.pow                // α **. ω
    !       mcode.factorial         mcode.binomial          // use !.
    sinθ    (a,w)=>Math.sin(w)      (a,w)=>a*Math.sin(w)
    cosθ    (a,w)=>Math.cos(w)      (a,w)=>a*Math.cos(w)
    tanθ    (a,w)=>Math.tan(w)      (a,w)=>a*Math.tan(w)
    asinθ   (a,w)=>Math.asin(w)     (a,w)=>a*Math.asin(w)
    acosθ   (a,w)=>Math.acos(w)     (a,w)=>a*Math.acos(w)
    atanθ   (a,w)=>Math.atan(w)     (a,w)=>a*Math.atan(w)
    atan2θ  'atan2'                 'atan2'
    ^       mcode.nyi               (a,w)=>a&&w             // logical AND   not binary
AND
    ∨       mcode.nyi               (a,w)=>a||w             // logical OR    not binary OR
    <       mcode.nyi               (a,w)=>+(a<w)           // less than
    >       mcode.nyi               (a,w)=>+(a>w)           // greater than
    ≤       mcode.nyi               (a,w)=>+(a<=w)          // less than or equals
    ≥       mcode.nyi               (a,w)=>+(a>=w)          // greater than or equals
    ∘       mcode.nyi               'outer'

        these are applied to each vector and matrix element
        +.× inner product uses rules of linear algebra
        use dot after ~ + - × / | ** to specify vectorized function    eg. 1 +. [ 2 3 ]
            nb. if no dot then the built-in scalar op is used
            eg. 'a' + [ 2 3 ]  + is string concat


    Built-in Scalar Operations
    symbol  JavaScript              comment
    ------  ----------              -------
    =       _                       // assignment
    ⍨       ===                     // exact equality test
    ==      _                       // built-in equals
    !=      _                       // built-in not equals
    !==     _                       // built-in not identical
    !       _                       // not (unary)
    ~       !                       // not (unary)                          reflexive
    ∨       ||                      // logical or   nb. not v is ∨          reflexive
    ^       &&                      // logical and  nb. shift 6 as exp      reflexive
    +       _                       // addition                             reflexive
    -       _                       // subtraction (unary)                  reflexive
    *       _                       // multiplication                       reflexive
    **      _                       // exponentiation                       reflexive
    %       _                       // modulus                              reflexive
    /       _                       // divide                               reflexive
    ??      _                       // nullish (undefined or null)
    <       _                       // less than
    >       _                       // greater than
    ≤       <=                      // less than or equals
    ≥       >=                      // greater than or equals

        _  means no change to symbol in this table
```

```
    reflexive  means do operation, then do assignment    eg. i += 2  means i=i+2
    ++ and -- are reflexive increment and decrement     eg. i++  means i=i+1
    = is general assignment  ≒ is exact equality test like ===
    regexp: use /s for space, since / means divide


  Operators
  operators = [ ¨ ⌿ ⍀ ⍣ @ ]
              ¨ mcode.each
              ⌿ mcode.reduce
              ⍣ mcode.power

    operators are functions that operate on functions,  syntax  L function operator R

    0 ⌈ ⌿ [ 2.1 3.7 ]           // ⌈ max reduction of list      is 3.70000
        nb. Left arg 0 selects the function ⌈ max, not function ⌈ ceiling

    ⌈ [ 2.1 3.7 ]               // ceiling over list            is [ 3 4 ]
        nb. Each operator ¨ could be used, but ⌈ is already vectorized

    'ab' ≢.xy ¨ [ 'ab cd' 'ab ef' ]                             is [ xy cd xy ef ]

    ⊙.r = 'new'
    'ab' ≢.⊙.r ¨ [ 'ab cd' 'ab ef' ]                            is [ new cd new ef ]
        nb. for each element in ⍵ replace ab with shared variable r


Notes About the Core file

    asi:true suppresses some semicolon warnings from jshint

    Output from the mcode transpiler is collected as the core file executes,
    then the JavaScript code may be written to an output file, to be used
    as a cache.  Use the 'Regen' button in the IDE to do this.

    This core file defines many functions called 'primitives' used in mcode.

    The transpiler output is collected in local variable 'r'
    and the 'r' option is also used in mexec as

    r += ⍎.er `...`

    - Since core file is building up the mcode system as it progresses, the
    code in the core file is a hybrid of JavaScript and mcode.
    Purer mcode examples are best found in other files.

    - Backquotes ` are escaped as \` if contained inside a backquoted string.
    Strings in strings must be double escaped, as in \\` etc.

    - The test functions in the core file are -not- collected into 'r'
    and are -not- written into the output cache file.

    The test functions are part of the local shared object ⊙.test
    and they typically call the assert primitive ?= after it is
    defined.

    - When defining a new primitive it is necessary to
    use a second pass of ⍎ mexec, since the parser will not recognize
    the new primitive until the -next- invocation of ⍎

    - The primituve ⍎ mexec invokes the transpiler and JavaScript
```

function eval. It is used to execute an mcode script,
add primitives to the language, or test the mcode
transpiler or language.

- To source a file into mcode, the usual expression is:

```
{ ⍺ ⍵ } ⍞ 'file.mc.txt'
```

where { ⍺ ⍵ } is a lambda function called when the ⍞ io read
primitive completes.

In an async function declared as ∇.a we can write:

```
∇.a foo
    ⍺ ⍞ 'file.mc.txt'
```

since an 'await' is inserted after the file read

The default file read location is the local folder on the cloud
or desktop, depending on where mcode is runnning.

- To source a file with debug output, the usual expression is:

```
{ ⍺.de ⍵ } ⍞ 'file.mc.txt'
```

- The built-in primitive ⌸ alone invokes the JavaScript debugger,
if it is already open.

Syntax Highlighting for the ACE editor

The file ace/mode-mcode.js defines regular expressions that
cause syntax highlighting.

New primitives as they are defined can be added to the 'regex' line:

```
token: "entity",   // primitives
regex: /⍁|⍂|⍺|⍞|⍬|∈|⍷|⍸|⍴|⍳|↑|↓|¨|≠|⌸|⌺|\?\=|∆/,
        (incomplete list)
```

New constants or built-in operators can be added also:

```
token: "support",   // built-ins
regex: /\+|\-|\*|\/|=|≈|�××|\^|∨|~|!|≠|×|÷|<|≤|≥|>|\||\&|π|ė|⎓/,
        (incomplete list)
```

Terminology and Approach Explained from an APL point of view

mcode is a transpiler that mixes in with the underlying language.

'operators' refer to built-in scalar functions in the underlying language.
However, in APL, 'operators' are functions that take other
functions as arguments creating a 'derived function'

mcode does not support derived functions, although lambdas create a similar
result.

Tacit or point-free syntax is not supported. In APL this is written
as primitives grouped away from data with ()

```
    The APL axis operator [] is not currently supported.
    Instead 'modifiers' may be used, as in:

    input               output
    -----               ------
    ⊢.1 0               mcode.nyi(null,0,1)  // nb. ⍺ = 1  ⍵ = 0
```

sidebar: Data Routing in APL vs. Function Routing

```
    nb. APL is greatly concerned with 'data routing'

    For example an mcode expression:

        z goo y foo x

        transpiles to:

        goo(z,foo(y,x));

    tacit or point-free syntax and derived functions essentially group
    and route data via the construction of function trees.

    Many OOP languages are instead concerned with 'function routing'.

    For example:

    foo(y,z).goo(x)

        where goo and foo belong to a superclass or agree on a common
        calling mechanism.

    To implement function routing, the first function must return some
    type of wrapper object, and all the functions are typically
    member functions of this wrapper object.

    Since mcode -does support OOP and classes-, function routing -can be used-
    in mcode, by using the no space rule, eg.

        C.foo(y,z).goo(x)

        where goo() and foo() are both members of class C, and goo() returns
        a reference to C.  Note in this example the execution order will
        be foo() then goo().  Also, goo() is not directly given the result of
        of foo() and must retreive it, often from another 'result holding' member
        in class C.

        In mcode we could also write:

        z C.goo y C.foo x

        which transpiles to:

        C.goo(z,C.foo(y,x));


    Another example is the promise 'then' syntax in JavaScript and C++.

    A(...).then(B(...))

        where 'then' is a clause or function that has specific meaning,
```

```
        such as asynchronous or future invocation in JavaScript.

        In this case, the wrapper class for A is a Promise or
        a 'thenable' class in JavaScript.


Debugging Tips for the JavaScript Transpiler

    In the transpiler code, there are some useful debug functions:

    tr(x,y) and trn(x,y) show x = y on the console and session logs
    x is a string label, and y will be decoded to one level of nesting.

    trn adds a newline

    jstr(x) is short for JSON.stringify(x)


    Constructing an array or object for temporary display is a helpful technique:

    code                                output
    ----                                ------
    trn('θ ',[op,mod,L,R]);             θ  = [ "θ"  "t"  "atan2"  "1"  ]

    trn('θ ',{op:op,L:L,R:R});          θ  = { op "θ"  L "atan2"  R "1"  }


    In the transpiler code, many key variables and flags have an optional debug flags:

    if (d) trn(...)
        where d is a local debug flag

    The main components of the transpiler are:

        initialization, lexer, expression builder, parser, mexec evaluator, loader

    Setting Debug mode for Transpiler components

        mexec also accepts suboptions to enable debugging for:

        L lexer, E expression builder, P parser

        The 'd' option is required also for debugging.

        Example: ⍎.'deE' <code>         debug, execute, show parser output

        Tips:
            use short expressions, since a lot of output will be created
            test one module at a time: L or E or P

    To test expressions against the transpiler interactively:

        in the IDE session:

        ⍞.opts 'deE'
        <enter an expression>


    To test expressions or functions against the transpiler in a file:

        in the core file 'core', or a test file:
```

```
      ⍝.deE `
      0 ⊢ 1           // short expression or code to test
      `
      /// stop        // optional, useful to stop lex,expr,parsing and eval here
```

Short Term Todo

- print Guide to PDF, start on help system per symbol
- migrate APL/VK to mcode VIK
- calendar widget for CATS/mcode
- transpiler & core file cleanup

Project Ideas

- port to Raspberry Pi and develop audio suite for radio mangling
- integrations with Babylonjs or Threejs, then drone/UAV demos
- family or classification tree layout and data manager
- process flow creator and web viewer
- process animation generator and exporter, see draw.io, CodeLink
- tools for CGI content creation, eg. models, textures, exporter

[end of document]

- Dedication by David:

  - With appreciation to Warren Juran, who showed me how to think about computing projects.
  - In memory of Professor Donald B. McIntyre (1923-2009), Pomona College, who introduced me to APL.

[end]